Variations on the Pipe and Filter Architectural Style

Songsakdi Rongviriyapanish, Nicole Levy

Variations on the Pipe and Filter Architectural Style

S. Rongviriyapanish[1] and N. Lévy[2]

[1] LORIA
BP. 239, F-54506 Vandœuvre-lès-Nancy, France
rongviri@loria.fr
[2] PRISM, Université de Versailles St-Quentin
10, 12 Av. de l'Europe, 78140 Velizy, FRANCE
Nicole.Levy@prism.uvsq.fr

Abstract. Formalising Architectural styles has been claimed as making variations impossible. We demonstrate that in the contrary, formalising enables to vary a style and to control that all the variations are still compatible with the style. To do so, we formalise the Pipe and Filter style together with several variations. We then show how to use these variations to develop the specification of a convolution product. The different solutions developed are proved equivalent and compared. The formal specification language LOTOS will be used as Architectural Description Language. The comparisons and other validations will be performed using the LOTOS environment CADP.

Keywords

Formal Specification Development, Architectural Styles and Patterns, Software Architecture, LOTOS, ADL, Formal Verification.

1 Introduction

The specification of software architecture is now recognised as a separate step in software life cycle. Architectural styles and patterns are techniques for identifying, describing and analysing commonly occurring architectural problems and solutions [Me98]. They characterise designs in terms of the system components and the connectors that enable communication between components [AAG93].

An arising problem is how to represent styles in such a way that unambiguous criteria can be stated to decide whether a given design conforms to some style and how a style representation can help to develop concrete architectures.

Buschmann et al. [BMR+96] claim formal methods do not apply to patterns, because "Formalisms ... tend to describe particular issues very precisely, but do not allow for the variation that is inherently embedded into every pattern". Moreover, they remark that there is no formalism "suitable for describing the benefits and liabilities of a pattern".

In contrast to this opinion, we deem it to be worthwhile to attempt a formalisation of design patterns and architectural styles for the following reasons:

- A formal description of architectural style clarifies the ambiguous points of the informal description improving their comprehensibility;
- Designers are provided with criteria to decide on the applicability of a certain pattern or style. For example, the definition of relations between patterns such as specialisation [LLM98] enable to compare them;

- Instantiation of styles is made easier by making explicit the parameters or characteristics of the style;
- Concrete architectures may be subject to proofs, analyses, and other forms of validation, e.g., animation
- The development of tools is facilitated by formal descriptions [EGY97].

Moreover, a formal description of patterns and styles does not prevent their variation and evolution. We demonstrate it by specifying variations on the Pipe and Filter style.

Informal circle-and-line drawings have shown their limitations and today, formal languages are proposed to represent software architectures. New languages for architectural descriptions have been developed, but they are still in a maturing phase, and few are provided with tools [Cle96].

Instead of creating a new ADL (Architectural Description Language) [SG96], we use the formal description language LOTOS [BB87]. In [HL97] and [Tur96], the authors have demonstrated that LOTOS is a suitable language to express architectural designs and in particular to establish a formal semantics of the communication between the components constituting a design pattern. In addition, data transformations can be specified as LOTOS abstract data types operations. Besides providing a formal semantics, the use of LOTOS has the advantage that existing tools, such as CADP (Caesar/Aldebaran Distribution Package) [FGM+92], can be employed to analyse and animate instances of patterns. Furthermore, LOTOS is an ISO standard such that a widespread familiarity with the language can be assumed.

In this paper we present an approach to express architectural styles using LOTOS patterns. A style is formalised by general patterns describing the properties of its components, connectors and configuration. In addition, some specific patterns are given as variations on the style. The characteristics of a style are to be considered as its parameters. They are the typed and constrained variables introduced in the patterns. We show that specifying a system according to a style becomes straightforward. It consists in instantiating correctly the characteristics. Our approach is illustrated by the formalisation of the Pipe&Filter style. This style and its variations is then used to develop different equivalent specifications of a convolution product. These specifications are analysed and compared one each other using the tool CADP.

In Section 2, the general approach followed to formalise architectural styles is presented. The approach is illustrated in Section 3 by characterising the architectural style Pipe&Filter. Several variations of filters and pipes are considered in Section 3.4. In Section 4, we present the case study of the convolution product. The concluding section discusses our approach in the context of related work. A summary of the LOTOS used in this paper is presented in the appendix.

2 Formalising Architectural Styles

2.1 What is in a style ?

Following the definition of the architectural building constructs [NR97], a style is described in four parts: (i) components and connectors, (ii) architectural configuration, (iii) architectural constraints and (iv) components and connectors variants.

- Components are units of computation or data stores. Their interface is a set of interaction points with either connectors or the environment.

- Connectors are architectural building blocks used to model interactions between components. The interface of a connector specifies the interaction points with the components attached to it.
- An architectural configuration is a connected graph of components and connectors describing the architectural structure. These informations are needed to ensure that components are correctly connected, that their interfaces match, that the connectors enable proper communications, and that their combined semantics results in the desired behaviour.
- Architectural constraints are the properties a specification must verify in order to be of the style. These constraints can be checked by any specification, without any indication of its development process.
- Variations of components and connectors describe some specific solutions.

The characteristics of the style describe its attributes. They can be considered as its parameters: what has to be given by the specifier in order to define an architecture. Characteristics are given for the components, the connectors and the configuration.

2.2 Architectural Styles in LOTOS

In this paper, we use the formal description language LOTOS [BB87] and its tool package CADP (Caesar/Aldebaran Distribution Package) [FGM+92]. LOTOS [BB87,LOT87] is a formal specification language developed to specify open distributed systems. LOTOS has two separated parts:

- the control part based on the process algebra approach for concurrency, combining features of CCS[Mil80] and CSP[Hoa85]. The semantics is defined in terms of labelled transitions systems.
- the data part for the description of data structures using algebraic abstract data types [GH78]. The ACT-One specification language [EM85] is used with conditional equations and an initial semantics. Data types are used for describing process parameters and values exchanged by the processes.

In the appendix, a summary of LOTOS used in this paper is presented.

A design description expressed in LOTOS consists of two parts. The global behaviour part describes the overall behaviour of the design, i.e., the configuration. The local definitions part contains the definition of the processes involved in the behaviour part and the definitions of the algebraic abstract data types introduced. The syntactic structure of a design description is as follows:

```
behavior
      behav_expr
where
      local_def_list
```

The basic ideas underlying our formalisation of software architecture descriptions are:

- The components and the connectors of an architecture are modelled as LOTOS processes and the requirements corresponding to a style will be defined by LOTOS patterns with variables;
- The architectural configuration of a style will defined by a LOTOS communication pattern;

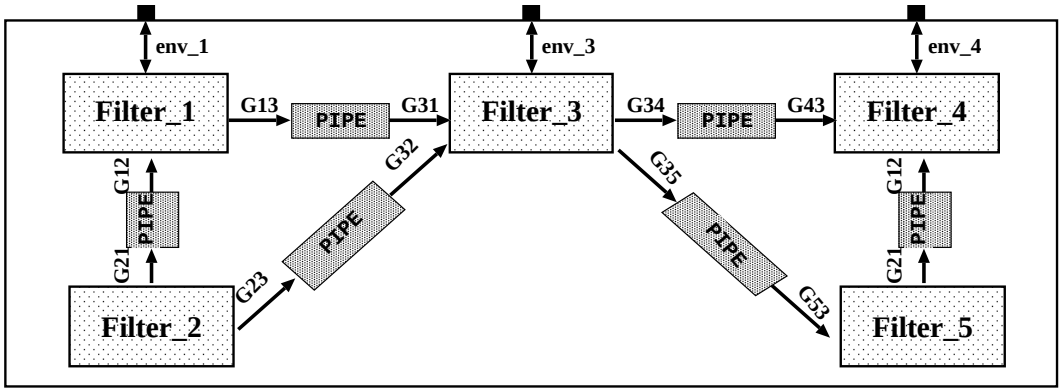

Fig. 1. A Pipe&Filter Architecture

- The characteristics of the style will be the list of variables introduced in the patterns. These variables are typed and constrained. The types represent parts of the specification. They are the ones of the nodes of the abstract syntax tree representing the specification. In the following, the variables will be written in italics.
- To obtain a concrete architecture, the specifier will have to give a value to these variables.

Using patterns present the advantage of providing guidelines to develop a system, ensuring that it will conform by construction to a given style.

3 Formalising the Pipe&Filter Style

The characteristics of Pipe&Filter style are the following [GS93]:

> "In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, [...] Components are termed "filters". The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence connectors are termed "pipes". [...] filters must be independent entities: in particular, they should not share state with other filters. "

Figure 1 shows an example of a Pipe&Filter architecture. A filter (such as `Filter_3`) may have several incoming and several outgoing pipes. Cycles are allowed.

3.1 Components and Connectors

All components are called **Filters** and all the connectors **Pipes**.

Filters A filter processes a local and incremental data transformation on the data received on its input gates and sends the results on its output gates. These results will be sent either to other filters via pipes or to the environment. A filter will perform three operations: receive the data on its input gates, compute an operation which will result in a several results that will be sent onto the output gates. A filter may have a local memory or a state. It is characterised by:

- the names of all its input and output gates *gate_list_IN* and *gate_list_OUT*,

– for each gate, the data type T_i of the values received or sent there,
– the data type T of its local state and
– the operation computed F whose signature is

$$F: T, T\_in\_1, \ldots T\_in\_n \rightarrow < T, T\_out\_1, \ldots T\_out\_m >$$

– the predicates $pred_{j \in 1..m}$ used as guard for the results emission.

Its behaviour can be described by the following minimal and abstract process:

```
process FILTER [gate_list_IN, gate_list_OUT] (val: T) : noexit :=
     (  gate_IN_1 ? v1 : T_in_1; exit(v1, any T_in_2, ... any T_in_n)
     ||| ...
     ||| gate_IN_n ? vn : T_in_n; exit(any T_in_1, ... any T_in_n-1, vn))
   >> accept v1: T_in_1, ... vn: T_in_n in
        exit (F(val, v1, ... vn))
   >> accept val': T, w1: T_out_1, ... wm: T_out_m in
     (   [pred1] -> ( gate_OUT_1 ! w1 ; exit ) [] [not(pred1)] ->  exit
     ||| ...
     ||| [predm] -> ( gate_OUT_1 ! w1 ; exit ) [] [not(predm)] ->  exit )
   >> FILTER [gate_list_IN, gate_list_OUT] (val')
endproc
```

where

$gate\_list\_IN = gate\_IN\_1,\ldots gate\_IN\_n$

$gate\_list\_OUT = gate\_OUT\_1,\ldots gate\_OUT\_m$

Let us note that the behaviour can be decomposed into three successive behaviours:

– A data reception behaviour on the n input gates in parallel and independently:

```
data_reception_behaviour =
       (  gate_IN_1 ? v1 : T_in_1; exit(v1, any T_in_2, ... any T_in_n)
       ||| ...
       ||| gate_IN_n ? vn : T_in_n; exit(any T_in_1, ... any T_in_n-1, vn))
```

– A computation behaviour where the operation F is called:

```
computation_behaviour includes
            F(val, v1, ... vn)
```

– A result transmission behaviour on the m output gates in parallel and independently:

```
result_transmission_behaviour =
    [pred1] -> ( gate_OUT_1 ! w1 ; exit ) [] [not(pred1)] ->  exit
                ||| ...
    [predm] -> ( gate_OUT_m ! wm ; exit ) [] [not(predm)] ->  exit
```

Pipes A pipe has two gates: an input gate where it receives a stream of data and an output gate where it sends it out, without any transformation and in the same order. A pipe is characterised by the names of both its input and output gates $gate\_IN$ and $gate\_OUT$ and the data type T of the values received and sent. Its behaviour can be described by the following minimal and abstract process:

```
process PIPE [gate_IN, gate_OUT]: noexit :=
   gate_IN ? x : T;
   gate_OUT ! x ;
   PIPE [gate_IN, gate_OUT]
endproc
```

3.2 Architectural Configuration

A system of Pipe&Filter style is characterised by the following attributes:

- the list of the names of the p filters $(FILTER\_i)_{i \in 1..p}$ and for each filter, its effective gates $gate\_list\_IN\_i$ and $gate\_list\_OUT\_i$;
- for each filter $FILTER\_i$, its characteristics as seen in section 3.1;
- the list of the names of the q pipes $(PIPE\_j)_{j \in 1..q}$ and for each pipe, its effective gates $gate\_IN\_j$ and $gate\_OUT\_j$;
- for each pipe $PIPE\_j$, its characteristics as seen in section 3.1;
- the list of the system external gates Ext_Gate_list.

The global configuration can be described by the following minimal and abstract behaviour expression:

```
hide all_gates_IN, all_gates_OUT in
  ((     FILTER_1 [gate_list_IN_1, gate_list_OUT_1] (val_1: T_1)
     ||| ...
     ||| FILTER_p [gate_list_IN_p, gate_list_OUT_p] (val_p: T_p))
  |[ all_gates_IN, all_gates_OUT ]|
   ( PIPE_1 [gate_IN_1, gate_OUT_1]
     |||
     ||| PIPE_q [gate_IN_q, gate_OUT_q]))
```

where

all_gates_IN, all_gates_OUT denote the communicating gates between the filters and the pipes. Each pipe input gate is a filter output gate, and each pipe output gate is a filter input gate:

$all\_gates\_IN = \cup_{i=1}^{q} gate\_IN\_i$

$all\_gates\_OUT = \cup_{i=1}^{q} gate\_OUT\_i$

$\forall j \in 1..q \ \exists! \ i \in 1..p \bullet gate\_IN\_j \in gate\_list\_OUT\_i$

$\forall j \in 1..q \ \exists! \ i \in 1..p \bullet gate\_OUT\_j \in gate\_list\_IN\_i$

In addition, the configuration must satisfy the following constraints:

1. The system communicates with its environment via the filters (the pipes do not communicate with the environment). As a consequence, the external gate list is equal to the union of filters gates not being a pipe gate):
   $Ext\_Gate\_list = (\cup_{i=1}^{p} gate\_list\_IN\_i \ \cup \ \cup_{i=1}^{p} gate\_list\_OUT\_i) \setminus (\cup_{j=1}^{q} gate\_IN\_j \ \cup \ \cup_{j=1}^{q} gate\_OUT\_j)$
2. Filters do not communicate directly, they do not share gates :
   $\forall i, i' \in 1..p : i \neq i' \Rightarrow (gate\_list\_IN\_i \cup gate\_list\_OUT\_i) \cap (gate\_list\_IN\_i' \cup gate\_list\_OUT\_i') = \varnothing$
3. Pipes do not communicate directly, they do not share gates :
   $\forall j, j' \in 1..q : j \neq j' \Rightarrow \{gate\_IN\_j, gate\_OUT\_j\} \cap \{gate\_IN\_j', gate\_OUT\_j'\} = \varnothing$

3.3 Architectural Constraints

The architectural constraints of the style Pipe&Filter describe properties to be satisfied by any system in order to be of the Pipe&Filter style. They are formalised with first order logic formulas. The considered system is a LOTOS specification neither defined with the above patterns nor by instantiation of the above variables. But the properties to be proved are the same as the ones constraining the characteristic variables.

The system is denoted by two variables: `beh` representing the global behaviour and `local_def` representing the definition of the processes introduced in `beh`. The constraints will concern the whole specification or just part of it. We suppose that the following functions are defined [1].

Ext_Gates:	*BEHAVIOR*	$\rightarrow$	*GATE_LIST*	List of the external gates (not hidden) of the behaviour
Proc_Calls:	*BEHAVIOR*	$\rightarrow$	*PROC_CALL_LIST*	List of the process call expressions in a behaviour
Name:	*PROC_CALL*	$\rightarrow$	*IDENT*	Name of the process called in a process call expression
Gates:	*PROC_CALL*	$\rightarrow$	*GATE_LIST*	Gates mentioned in a process call expression
Proc_Def:	*LOCAL_DEF*, *IDENT*	$\rightarrow$	*PROC_DEF*	Returns the definition of a named process from the local definitions

The types represent parts of the specification. They are the ones of the nodes of the abstract syntax tree representing the specification. In addition, we suppose one attribute *Archi* associated to each process definition. For example, a process definition denoted by the variable f representing a filter will have as *Archi* attribute `Filter`, denoted: *Archi(f)* `= Filter`. Finally and to simplify the notation, we define the sets *filters* and *pipes* denoting the filters and the pipes of a system as follows:

$$\begin{aligned} filters(beh, local\_def) = \{&f \in Proc\_Calls(beh) \mid \\ &Archi(Proc\_Def(local\_def, Name(f))) = \texttt{Filter}\} \\ pipes(beh, local\_def) = \{&p \in Proc\_Calls(beh) \mid \\ &Archi(Proc\_Def(local\_def, Name(p))) = \texttt{Pipe}\} \end{aligned}$$

Constraints:

1. The system communicates with its environment via the filters (the pipes do not communicate with the environment):
   $\forall\, p \in pipes(beh, local\_def) \bullet Gates(p) \cap Ext\_Gates(beh) = \varnothing$
2. Data is carried between filters only via pipes (filters do not share gates):
   $\forall f1, f2 \in filters(beh, local\_def) \bullet Gates(f1) \cap Gates(f2) = \varnothing$
3. Each pipe has two gates. It links an output gate of a filter to an input gate of another filter:
   $\forall\, p \in pipes(beh, local\_def)\, \exists\, g_{in}, g_{out} \bullet Gates(p) = (g_{in}, g_{out}) \;\wedge$
   $(\exists f_{in}, f_{out} \in filters(beh, local\_def) \bullet g_{in} \in Gates(f_{in}) \;\wedge\; g_{out} \in Gates(f_{out}) \wedge f_{in} \neq f_{out})$
4. All the pipes are independent one each other (they do not share gates).
   $\forall p1, p2 \in pipes(beh, local\_def) \bullet Gates(p1) \cap Gates(p2) = \varnothing$
5. All filters must conform to the behaviour detailed in section 3.1 being equivalent with respect to the safety equivalence [FM91] to the instantiated pattern defining the filters behaviour.
6. All pipes must conform to the behaviour detailed in section 3.1 being equivalent with respect to the safety equivalence [FM91] to the instantiated pattern defining the pipes behaviour.

3.4 Variations on the Pipe&Filter Style

The variations on filters and pipes represent some specific solutions. They are equivalent with respect to the safety equivalence to the patterns given in section 3.1.

[1] for example on the abstract syntax tree of the specification.

Variations on Pipes The general pipe pattern describes the expected pipe behaviour: to receive a message on its input gate and to send it on its output gate. It is possible to decouple these two behaviours, for example to enable several successive receptions before a sending. To do so, the pipe must have a local buffer, managed as a file, to store the received messages waiting to be output. This buffer can either be bounded or not. To model such pipes, we define algebraically the abstract data types `BUF` and `BOUNDED_BUF` provided with the operations `init`, `put`, `head`, `tail`, `empty` and `full`. The definitions of these types are given in Table 1 and 2.

A bounded buffer is composed of an integer, representing its size, and an unbounded buffer containing the elements. Therefore, most of its operations can be described as extensions of the unbounded buffer operations. It is the case for `put`, `tail`, `size`, `head`, and `empty`. To define them, the buffer part is taken by the projection operation: `buf_of_bounded` and the operation of an unbounded buffer is applied to it. For example, the operation `empty` is defined as follows: `empty(A_bounded) = empty(buf_of_bounded(A_bounded))` where `A_bounded` is a bounded buffer. This equation says that a bounded buffer is empty if its buffer is empty.

We obtain two pipe variations by instantiating them with different buffer types. Their behaviours correspond to the following patterns:

Bounded Pipe

```
process PIPE [gate_IN, gate_OUT] ( A_Bounded: BOUNDED_BUF ) : noexit:=
    [not (full( A_Bounded ))] ->
      gate_IN ? x: T ;
      PIPE [gate_IN, gate_OUT] (put( x, A_Bounded ))
  []
    [not (empty( A_Bounded ))] ->
      gate_OUT ! head( A_Bounded ) ;
      PIPE [gate_IN, gate_OUT] (tail( A_Bounded ))
endproc
```

Unbounded Pipe

```
process PIPE [gate_IN, gate_OUT] ( An_Unbounded: BUF ) : noexit :=
      gate_IN ? x: T ;
      PIPE [gate_IN, gate_OUT] (put( x, An_Unbounded ))
  []
    [not (empty( An_Unbounded ))] ->
      gate_OUT ! head( An_Unbounded ) ;
      PIPE [gate_IN, gate_OUT] (tail( An_Unbounded ))
endproc
```

Variations on Filters As seen in the general pattern, a filter performs three behaviours: a data reception, a computation and a result transmission. These behaviours can be executed either sequentially as in the general pattern, or in parallel: for example, the filter could start receiving new inputs before having terminated its computation. We will call this behaviour, a **non-blocking filter**. In this case, the three behaviours are composed in parallel, synchronising on two internal (and hidden) gates *int_gate1* and *int_gate2*. The reception process synchronises its inputs with the computation process on *int_gate1* whereas the computation process synchronises its results with the transmission process on *int_gate2*. In contrast, the

Table 1. ADT definition of Unbounded buffer

Unbounded Buffer

```
library X_BOOLEAN, NATURAL, EXP endlib
type SYMB_EXP_UNBOUNDED_BUFFER is BOOLEAN, EXP, NATURAL
sorts BUF
opns
    init     :              -> BUF
    put      : EXP, BUF     -> BUF
    tail     : BUF          -> BUF
    head     : BUF          -> EXP
    empty    : BUF          -> BOOL
    size     : BUF          -> NAT
    _nequal_ : BUF, BUF     -> BOOL
    _equal_  : BUF, BUF     -> BOOL
eqns forall e1,e2:EXP,  An_Unbounded1, An_Unbounded2:BUF
  ofsort BOOL
    empty(init)          = true;
    empty(put(e1, An_Unbounded1)) = false;
   (An_Unbounded1 nequal An_Unbounded2) = not(An_Unbounded1 equal An_Unbounded2);
    init equal init = true ;
    init equal put(e1, An_Unbounded1) = false ;
    e1 ne e2 => put(e1, An_Unbounded1) equal put(e2, An_Unbounded2) = false ;
    put(e1, An_Unbounded1) equal put(e1, An_Unbounded2)
         = ( An_Unbounded1 equal An_Unbounded2 ) ;

   (An_Unbounded1 equal An_Unbounded2) = (An_Unbounded2 equal An_Unbounded1)
  ofsort BUF
    tail(init) = init ;
    tail(put(e1, init)) = init ;
    An_Unbounded1 nequal init =>
         tail(put(e1, An_Unbounded1))= put(e1, tail(An_Unbounded1));
  ofsort NAT
     size(init) = 0 ;
     size(put(e1, An_Unbounded1)) = Succ(size(An_Unbounded1))

  ofsort EXP
     head(put(e1,init)) = e1;
     head(init) = 0 ;

     An_Unbounded1 nequal init =>
         head(put(e1, An_Unbounded1)) = head(An_Unbounded1)
endtype
```

Table 2. ADT definition of bounded buffer

Bounded Buffer

```
library X_BOOLEAN, NATURAL, EXP endlib
type SYMB_EXP_BOUNDED_BUFFER is BOOLEAN, EXP, NATURAL,
                                SYMB_EXP_UNBOUNDED_BUFFER
sorts  BOUNDED_BUF
opns
      init      : NAT, BUF                      -> BOUNDED_BUF
      put       : EXP, BOUNDED_BUF              -> BOUNDED_BUF
      tail      : BOUNDED_BUF                   -> BOUNDED_BUF
      head      : BOUNDED_BUF                   -> EXP
      empty     : BOUNDED_BUF                   -> BOOL
      bound     : BOUNDED_BUF                   -> NAT
      size      : BOUNDED_BUF                   -> NAT
     _nequal_   : BOUNDED_BUF, BOUNDED_BUF      -> BOOL
     _equal_    : BOUNDED_BUF, BOUNDED_BUF      -> BOOL
      full      : BOUNDED_BUF                   -> BOOL
 buf_of_bounded : BOUNDED_BUF                   -> BUF

eqns forall e1:EXP, An_Unbounded:BUF, n:NAT, A_Bounded1, A_Bounded2: BOUNDED_BUF
  ofsort BOOL
   (A_Bounded1 nequal A_Bounded2) = not(A_Bounded1 equal A_Bounded2) ;

   (A_Bounded1 equal A_Bounded2)
        = (buf_of_bounded(A_Bounded1) equal buf_of_bounded(A_Bounded2)) ;

    full(A_Bounded1) = (bound(A_Bounded1) eq size(A_Bounded1) ) ;
    empty(A_Bounded1) = empty(buf_of_bounded(A_Bounded1) )

  ofsort BUF
    buf_of_bounded(init(n, An_Unbounded)) =  An_Unbounded

  ofsort BOUNDED_BUF
    tail(A_Bounded1) = init(bound(A_Bounded1), tail(buf_of_bounded(A_Bounded1) ) ) ;

   (bound(A_Bounded1) gt size(A_Bounded1)) =>
     put(e1,A_Bounded1)= init(bound(A_Bounded1),put(e1,buf_of_bounded(A_Bounded1)));

   (bound(A_Bounded1) le size(A_Bounded1)) => put(e1,A_Bounded1) = A_Bounded1

  ofsort NAT
     bound( init(n, An_Unbounded) ) = n ;
     size(A_Bounded1) = size( buf_of_bounded(A_Bounded1) )

  ofsort EXP
     head(A_Bounded1) = head(buf_of_bounded(A_Bounded1))
endtype
```

general pattern filter given in section 3.1 will be called a **blocking filter**. Both patterns of the non-blocking and the blocking filters are given below:

Non Blocking Filter

```
process FILTER [gate_list_IN, gate_list_OUT] (val: T) : noexit :=
    hide int_gate1,int_gate2 in
      ( Reception [gate_list_IN, int_gate1]
          |[int_gate1]|
        Computation [int_gate1, int_gate2]( val )
          |[int_gate2]|
        Transmission [gate_list_OUT,int_gate2] )
where
    process Reception [gate_list_IN, int_gate1] : noexit :=
           data_reception_behaviour
           >> accept v1: T_in_1,... vn: T_in_n in
             ( int_gate1 !v1 ... !vn ;
               Reception [gate_list_IN, int_gate1])
    endproc
    process Computation [int_gate1,int_gate2] ( val: T ) : noexit :=
           computation_behaviour
        >> accept val': T in
           Computation [int_gate1,int_gate2] ( val')
    endproc
    process Transmission [gate_list_OUT,int_gate2]: noexit :=
         int_gate2 ?val': T ?w1:T_out_1 , ... ?wm:T_out_m ;
         result_transmission_behaviour
        >> Transmission [gate_list_OUT,int_gate2]
    endproc
endproc
```

Blocking Filter

```
process FILTER [gate_list_IN, gate_list_OUT] (val: T) : noexit :=
           Reception [gate_list_IN]
    >> accept v1: T_in_1,... vn: T_in_n in
          (Computation( val, v1, ... vn )
    >> accept val': T, w1: T_out_1, ... wm: T_out_m in
          (Transmission [gate_list_OUT]( w1, ... wm )
           >> FILTER [gate_list_IN, gate_list_OUT] (val') )
where
    process Reception [gate_list_IN]
                    : exit( val: T, v1: T_in_1,... vn: T_in_n ) :=
        data_reception_behaviour
    endproc
    process Computation( val: T, v1: T_in_1,... vn: T_in_n ) :
                         exit ( val': T, w1: T_out_1, ... wm: T_out_m  ) :=
        computation_behaviour
    endproc
    process Transmission [gate_list_OUT] ( w1: T_out_1,... wm: T_out_m):
                         exit :=
        result_transmission_behaviour
    endproc
endproc
```

The three variables *data_reception_behaviour*, *computation_behaviour* and *result_transmission_behaviour* denote the behaviours defined in section 3.1.

4 Case Study: a Convolution Product

A convolution product defines a sequence of values Y_i by summing up the product of n values: $W_1, \cdots, W_n$ with a sequence of values X_i. In this example, we will just consider three values ($n = 3$).

$$Y_i = \sum_{1 \leq j \leq n} (W_j * X_{i+j-1}) \tag{1}$$

A systolic network [Kun82,Gar89] with three identical and connected cells running the same algorithm can be used to solve this problem. As shown in Figure 2, a systolic network is typically of Pipe&Filter style.

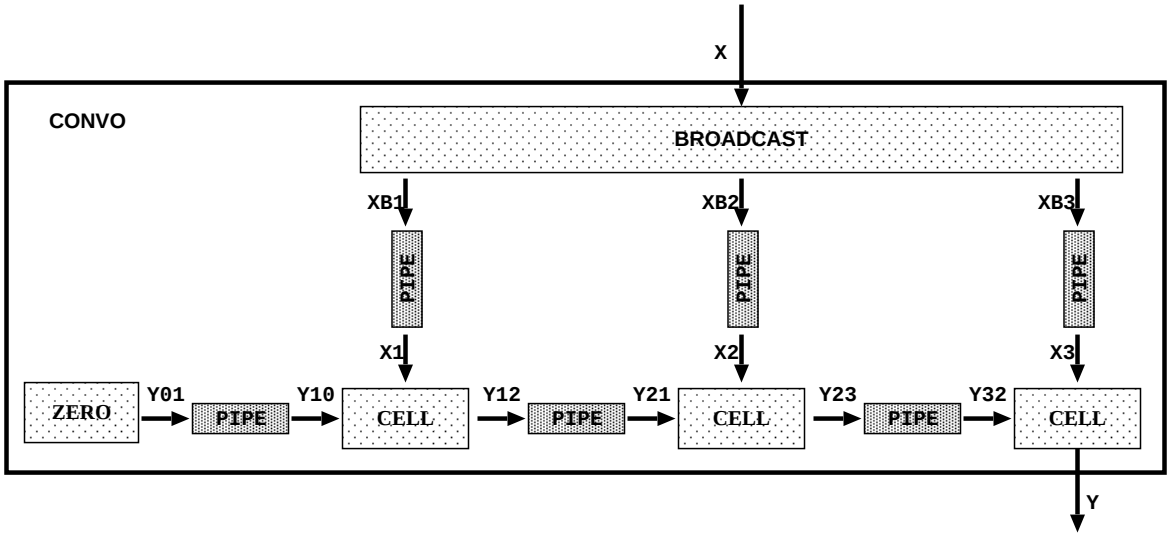

Fig. 2. Convolution product using systolic solution

Following a top-down approach, our development has three steps:

- *step(i)*: identification of the interface of the system with its environment (instantiation of *Ext_Gate_list*) and type of the values exchanged with it.
- *step(ii)*: identification of the components and connectors and for each of them definition of their interface (instantiation of the list of filters names $(FILTER\_i)_i$ with their effective gates *gate_list_IN_i* and *gate_list_OUT_i*, the data type T of the local state and the operation computed F and the list of pipes names $(PIPE\_i)_i$ with their effective gates *gate_IN_i* and *gate_OUT_i*);
- *step(iii)* definition of the behaviour of each filter and pipe by choosing a pattern.

At each step, the constraints given in the style definition must be satisfied.

step(i) Two external gates are required: an input gate `X` to receive the sequence $(X)_i$ and an output gate `Y` to send out the sequence of results $(Y)_j$. The external gate list represented by the variable *Ext_Gate_list* is equal to (`X, Y`). On both `X` and `Y`, the values are of a symbolic expression type denoted `EXP`.

```
specification Convo[X, Y] : noexit
behaviour
   behav_expr
where
   local_def_list
endspec
```

step(ii) As shown in Figure 2, we identify three kinds of filters: `BROADCAST`, `ZERO`, and `CELL`. For each one, their characteristics are the following:

- `BROADCAST` receives the sequence of $(X)_i$ on its gate `X` and broadcasts its values to the cells sending them via pipes on gates `XB1, XB2, XB3`. Data on these gates `X, XB1, XB2, XB3` are of type `EXP`. `BROADCAST` is parameterised by the number of cells to which it must broadcast the values.
- `ZERO` initialises the computation by sending a zero to the first cell via a pipe on gate `Y01`.
- Each component `CELL` computes a product of an input with its own weight W_i and sums it with the current result received. The new result is sent out. Therefore, a cell has two input and an output gates. The first cell receives the value zero on gate `Y10` and the X_i on gate `X1`, it sends the results on gate `X12`. The second cell receives the current result on gate `Y21` and the X_i on gate `X2`, it sends the results on gate `X23`. The last cell receives the current result on gate `Y32` and the X_i on gate `X3`, it sends its result to the environment on gate `Y`. Each cell is parameterised by its weight W_i.

All the filters are connected by pipes called `PIPE`. Table 3 summarises the characteristics of the components and connectors of the convolution product according to the Pipe&Filter style. From these characteristics, we instantiate the pattern of global configuration given in Section 3.2.

Architectural Configuration

```
behav_expr =
      hide XB1, XB2, XB3, X1, X2, X3, Y01, Y10, Y12, Y21, Y23, Y32 in
       ((     BROADCAST [X, XB1, XB2, XB3](3)
          ||| ZERO [Y01]
          ||| CELL [X1, Y10, Y12] (W1)
          ||| CELL [X2, Y21, Y23] (W2)
          ||| CELL [X3, Y32, Y]   (W3) )
        |[ XB1, XB2, XB3, X1, X2, X3, Y01, Y10, Y12, Y21, Y23, Y32 ]|
        (     PIPE [XB1, X1]
          ||| PIPE [XB2, X2]
          ||| PIPE [XB3, X3]
          ||| PIPE [Y01, Y10]
          ||| PIPE [Y12, Y21]
          ||| PIPE [Y23, Y32] ))
```

Let us note that, with p the number of filters equal to 5 and q the number of pipes equal to 6, the constraints number 1, 2 and 3 concerning the configuration and the conditions on global configuration given in section 3.2 are satisfied.

Table 3. Summary of characteristics of the convolution product following the Pipe&Filter style

Component	*gate_list_IN*: data type	*gate_list_OUT*: data type	data type of local state
(Filter) BROADCAST	X : EXP	X1 : EXP, X2 : EXP, X3 : EXP	NAT(3)
(Filter) ZERO	-	Y01 : EXP	-
(Filter) CELL	X1 : EXP, Y10 : EXP	Y12 : EXP	EXP(W_1)
(Filter) CELL	X2 : EXP, Y21 : EXP	Y23 : EXP	EXP(W_2)
(Filter) CELL	X3:EXP, Y32:EXP	Y:EXP	EXP(W_3)

Connector	*gate_IN*: data type	*gate_OUT*: data type
PIPE	Y01 : EXP	Y10 : EXP
PIPE	Y12 : EXP	Y21 : EXP
PIPE	Y23 : EXP	Y32 : EXP
PIPE	XB1 : EXP	X1 : EXP
PIPE	XB2 : EXP	X2 : EXP
PIPE	XB3 : EXP	X3 : EXP

List of the external gates : *Ext_Gate_list* = (X, Y)

step(iii$_1$*)* Filters Behaviour.

- BROADCAST sends to the three cells the value received. Therefore, the computation is the identity. Concerning the emission, during the initialisation, BROADCAST has a special behaviour[2]: X_1 is just sent to the first cell, X_2 to the two first cells and the other $X_i (i \geq 3)$ to every cell. This behaviour is obtained by guarding the emissions. Given D a delay that is initially equal to K the number of cells, the delay D is decreased for each iteration until zero. BROADCAST will send the values to a number of cells depending on the delay. If the delay D is zero, then it sends the values to all the cell. Otherwise, it will send them to all the cells except the last D ones. For example in our case study with three cells, if the delay D is equal to 2, a value will be sent to the first cells and not to the last two cells. BROADCAST can be specified either as a blocking or a non-blocking filter. Their specifications are given below.
- ZERO is a degenerated kind of filter: it has no input, it computes a constant 0 and sends it out on one gate. We instantiate the general pattern with $n = 0$, $m = 1$ and $F = 0$. It can be simplified with neither data reception nor computation behaviours.

```
process ZERO [Y01] : noexit :=
   Y01 !(0 of EXP);
   ZERO [Y01]
endproc
```

- A CELL adds to the current result the product of an input X_i with its own weight W_i and sends out the result. We can specify it either as a blocking or a non-blocking filter. Its

[2] This initialisation phase is typical in a pipeline system. A number of delays equal to the number of filters is required by data flow to attain the rightmost filter:
$Y_1 = W_1 * X_1 + W_2 * X_2 + W_3 * X_3$
$Y_2 = W_1 * X_2 + W_2 * X_3 + W_3 * X_4$
$Y_3 = W_1 * X_3 + W_2 * X_4 + W_3 * X_5$
$Y_4 = W_1 * X_4 + W_2 * X_5 + W_3 * X_6$
...

BROADCAST defined as a Blocking Filter

```
process BROADCAST [X, X1, X2, X3]( K: NAT ): noexit
        :=
          Reception [X] >> accept x1: EXP in ( Computation( x1, K )
            >> accept y1: EXP, y2: EXP, y3: EXP, K1: NAT in
              ( Emission [X1, X2, X3]( y1, y2, y3, K1 )
                >> BROADCAST [X, X1, X2, X3]( K1 ) ) )
where
     process Reception [X] : exit (EXP)
          := ( X ?x1: EXP; exit(x1) )
     endproc
     process Computation ( x1: EXP, K: NAT ) : exit ( EXP, EXP, EXP, NAT )
          :=  [K eq 0] -> exit ( x1, x1, x1, K )
            []
              [K gt 0] -> exit ( x1, x1, x1, (K - 1))
     endproc
     process Emission [X1, X2, X3] ( y1: EXP, y2: EXP, y3: EXP, K1: NAT ):exit
         :=
       ( ( ([K <= 2] -> ( X1 !y1 ; exit )) [] ([not (K <= 2)] -> exit) )
      ||| ( ([K <= 1] -> ( X2 !y2 ; exit )) [] ([not (K <= 1)] -> exit) )
      ||| ( ([K eq 0] -> ( X3 !y3 ; exit )) [] ([not (K eq 0)] -> exit) ) )
     endproc
endproc
```

BROADCAST defined as a Non-blocking Filter

```
process BROADCAST [X, X1, X2, X3]( K: NAT ): noexit
   :=
      hide int1, int2 in (
          reception [X, int1]
             |[int1]| Computation[int1, int2]( K )
             |[int2]|Emission [int2, X1, X2, X3] )
where
      process Reception [X, int1] : noexit
          :=
              X ?x1: EXP; exit(x1)
              >> accept x1: EXP in
                   ( int1 !x1; Reception [X, int1] )
      endproc
      process Computation [int1, int2] ( K: NAT ) : noexit
          :=  [K eq 0] -> int1 ? y1: EXP;
                          int2 !y1 !y1 !y1 !K ;
                          Computation [int1, int2] ( K )
            []
              [K gt 0] -> int1 ? y1: EXP;
                          int2 !y1 !y1 !y1 !(K-1) ;
                          Computation [int1, int2] ( K-1 )
      endproc
      process Emission [int2, X1, X2, X3] : noexit
         :=
         int2 ?y1: EXP ?y2: EXP ?y3: EXP ?K: NAT ;
       ( ( ([K <= 2] -> ( X1 !y1 ; exit )) [] ([not (K <= 2)] -> exit) )
      ||| ( ([K <= 1] -> ( X2 !y2 ; exit )) [] ([not (K <= 1)] -> exit) )
      ||| ( ([K eq 0] -> ( X3 !y3 ; exit )) [] ([not (K eq 0)] -> exit) ) )
           >> Emission [int2, X1, X2, X3]
     endproc
endproc
```

computation is defined as follows:

F: EXP, EXP, EXP $\rightarrow$ *EXP*

$F(W_i,\ X_i,\ Y) = W_i \star X_i + Y$

The specification of CELL defined both as a blocking and a non-blocking filter are given below.

step(iii$_2$) Pipes Behaviour.
We have six pipes. Each of them can either be a simple, a bounded or an unbounded pipe. Provided given this choice and the data type EXP of the values passing through the pipe, the instantiation is systematic. Let us note that in both cases of bounded and unbounded pipes, in the architectural configuration, the buffers have to be initialised to init, for example: an unbounded buffer initialised by PIPE [XB1, X1](init of BUF) and a buffer with 3 places by PIPE [XB1, X1](init (3, init of BUF)).

In order to be able to verify the architectural constraints of the style, some annotations are required . The introduced processes are annotated with the attribute *Archi* equal to Pipe (for the process PIPE) or to Filter (for the processes BROADCAST, ZERO and CELL). The annotated specification satisfies all the architectural constraints given in section 3.3. It confirms the conformity of the specification to the Pipe&Filter style.

Results
The different solutions compared are constructed varying the kind of cells (blocking or non-blocking), of broadcast filter (blocking or non-blocking), of pipe (simple, bounded or unbounded). In addition, the data types used are or not compiled in C. The CADP tool (Caesar/Aldebaran Distribution Package) [FGM+92] is used to compare the different solutions obtained using style variations. All the solutions have been demonstrated equivalent with respect to the safety equivalence [FM91]. Table 4 compares the size of the labelled transitions systems (LTS) computed by the tool.

Cases (a) and (b) differ on the EXP data type definition. In (a), the specification includes the abstract data type definition while in (b), the data type is defined externally, directly in C. The LTS are the same.

Cases (c) and (d) compare bounded and unbounded pipes. In (c) the size of the buffer is 3 for pipes between BROADCAST and cells, 1 for the pipe between ZERO and the first cell. In case (d), the pipes are unbounded except for the one linking ZERO (an unbounded pipe makes it possible for ZERO to send infinitely its zeros, the LTS becoming infinite). Naturally, the (d) LTS is larger than the (c) one.

Cases (e), (f) and (g) introduce non-blocking filters. Again, this increases the LTSs.

5 Conclusion

We have presented, by means of the Pipe&Filter style, a methodology to formalise architectural styles. To specify is to describe a behaviour. Our formalisation proposes both general and specific patterns that can be directly used in order to develop an architecture. All the patterns corresponding to a component or to a connector are equivalent with respect to safety equivalence. The characteristics of the style, defined as typed and constrained variables, are to be considered as its parameters. Making them explicit, makes it straightforward to instantiate the style. The resulting specifications are by nature of the style, i.e. the architectural constraints are satisfied.

CELL defined as a Blocking Filter

```
process CELL [X_IN, Y_IN, Y_OUT] (W:EXP) : noexit :=
        Reception [X_IN, Y_IN]
           >> accept X:EXP, Y:EXP in  Computation (X, Y, W)
              >> accept Z: EXP in (
                     Emission [Y_OUT] (Z)
                     >> CELL [X_IN, Y_IN, Y_OUT] ( W ) )
where
    process Reception [IN1,IN2] : exit (EXP, EXP) :=
              ( IN1 ?X: EXP ; exit(X,any EXP)
                          |||
                IN2 ?Y: EXP ; exit(any EXP,Y)   )
              >> accept X, Y : EXP in
                   exit(X, Y)
    endproc
    process Computation (X, Y: EXP, W: EXP) : exit(EXP) :=
               exit (Y + (W * X))
    endproc
    process Emission [OUT1] (Z: EXP): exit :=
             ( OUT1 !Z ; exit )
    endproc
endproc
```

CELL defined as Non-Blocking Filter

```
process CELL [X_IN, Y_IN, Y_OUT] (W:EXP) : noexit :=
      hide IG1, IG2 in
             Reception   [IG1, X_IN, Y_IN]
        |[IG1]|
             Computation [IG1, IG2] (W)
        |[IG2]|
             Sending     [IG2, Y_OUT]
where
    process Reception [IG, IN1,IN2] : noexit :=
           ( IN1 ?X: EXP ; exit(X,any EXP)
                         |||
             IN2 ?Y: EXP ; exit(any EXP,Y)   )
           >> accept X, Y : EXP in
                IG !X !Y ;
                Reception [IG, IN1,IN2]
    endproc
    process Computation [IG1,IG2](W: EXP) : noexit :=
            IG1 ?X: EXP ? Y: EXP ;
'            IG2 !(Y + (W * X));
            Computation [IG1,IG2](W)
    endproc
    process Sending [IG, OUT1] : noexit :=
            IG   ? Z : EXP ;
           ( OUT1 !Z ; exit )
            >> Sending [IG, OUT1]
    endproc
endproc
```

Table 4. Comparing Variations of the Specifications

case	CELL	BROADCAST	data type	PIPE	buffer size	states	transitions
(a)	blocking	blocking	`EXP(ADT)`	simple	1	38694	160241
(b)	blocking	blocking	`EXP(C)`	simple	1	38694	160241
(c)	blocking	blocking	`EXP(C),BOUNDED_BUF(ADT)`	bounded	3	269311	1189813
(d)	blocking	blocking	`EXP(C),BUF(ADT)`	unbounded	-	331250	1466727
(e)	blocking	non-blocking	`EXP(C)`	simple	1	64188	285230
(f)	non-blocking	blocking	`EXP(C)`	simple	1	67756	315450
(g)	non-blocking	non-blocking	`EXP(C)`	simple	1	97798	469611

Formal descriptions of architectural styles and concrete architectural designs are important because only architectural descriptions with a formal semantics make it possible to precisely answer the questions stated by Clements [Cle96]: What are the components? How do they behave? What do the connections mean?

We are not the first one to formalise software architectures and styles. Ad hoc notations have been defined such as Wright[All97] or Rapide [DJL+95]. Several existing formal notations have also been used. Among others, we can cite CSP [AG94], Z [DG91,AAG93], Larch [CP97], graph grammars [Met96] and even LOTOS [Tur98]. One of the advantages of using an existing formalism, is the availability of environments or validation tools such as CADP we used. LOTOS, with its two parts: a process algebra and algebraic abstract data types, enable the behaviour together with data transformations to be specified.

Concerning the variations, there is not much existing work. The question is: how the different variations are related? We propose the safety equivalence. The refinement patterns defined in [MQ94] could be considered as variations patterns, seeing refinement as a kind of variation.

We are very much concerned with the development process. In the example, we have sketched a method for developing architectures introducing different steps. This approach is related to the *agendas* [Hei98].

Future work include the definition of the architectural styles as development operators of the PROPLANE environment [SL96]. In PROPLANE the development of a specification is defined as a sequence of steps in which each step maps some development state to the next one by the application of some operator. When using this tool, the specifier selects predefined operators in a library. Constraints are associated to the operators and verified at each step. This makes the development secure and easy.

References

[AAG93] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 18(5), pages 9–20. ACM Press, December 1993.

[AG94] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings 16th Int. Conf. on Software Engineering*. ACM Press, 1994.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, May 1997.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, a System of Patterns.* J. Wiley and Sons, Inc, 1996.

[Cle96] P. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 16–25, Schloss Velen, Germany, March 1996. IEEE.

[CP97] P. Ciancarini and W. Penzo. Validating a software architecture with respect to an architectural style. Technical Report UBLCS-97-7, University of Bologna (Italy). Department of Computer Science., URL:ftp://ftp.cs.unibo.it/pub/techreports/97-07.ps.gz, July 1997.

[DG91] D. Notkin D. Garlan. Formalizing design space: Implicit invocation mechanisms. In W. Toetenel S. Prehn, editor, *Proceedings of the 4th Annual Symposium: VDM '91*, volume 551 of *Lecture Notes in Computer Science*, pages 31–44. sv, October 1991. 1991.

[DJL+95] D. C. Luckham, J. J. Kenny, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, pages 336–355, April 1995.

[EGY97] A. Eden, J. Gil, and A. Yehudai. Automating application of design patterns. *Object-Oriented Programming*, 10(2), May 1997.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[FGM+92] J-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. A toolbox for the verification of lotos programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259. ACM, May 1992.

[FM91] Jean-Claude Fernandez and Laurent Mounier. A tool set for deciding behavioral equivalences. In *Proceedings of CONCUR'91 (Amsterdam, The Netherlands)*, August 1991.

[Gar89] H. Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de doctorat, Université Joseph Fourier (Grenoble), November 1989.

[GH78] JV. Guttag and JJ. Horning. The algebraic specification of abstract data tupes. *Acta Informatica*, 10:27–52, 1978.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Company*, 1, 1993.

[Hei98] M. Heisel. Agendas – a concept to guide software development activites. In R. N. Horspool, editor, *Proc. Systems Implementation 2000*, pages 19–32, London, 1998. Chapman & Hall.

[HL97] M. Heisel and N. Lévy. Using LOTOS patterns to characterize architectural styles. In *Proc. 7th International Joint Conference on the Theory and Practice of Software Development, (TAPSOFT'97-FASE)*, volume 1214 of *Lecture Notes in Computer Science*, Lille, France, April 1997.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[Kun82] H. T. Kung. Why systolic architectures ? *Computer*, 15(1):37–46, january 1982.

[LLM98] N. Levy, F. Losavio, and A. Matteo. Comparing architectural styles: Broker specializes mediator. In J. Magee and D. Perry editors, editors, *Proceedings of the Third International Software Architecture Workshop (ISAW 3)*, pages 93–96, Orlando, Florida, USA, Nov 1998. SIGSOFT, ACM Press.

[LOT87] ISO. LOTOS. A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Draft international standard 8807, International Organization for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva, 1987.

[Me98] J. Magee and D. Perry editors, editors. *Third International Software Architecture Workshop (ISAW 3)*, Orlando, Florida, USA, Nov 1998. SIGSOFT, ACM Press.

[Met96] D. Le Metayer. Software architecture styles as graph grammars. In *In Proc of the ACM SIGSOFT Symposium of the foundations of Software Engineering*, pages 15–23, 1996.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[MQ94] M. Moriconi and X. Qian. Correctness and composition of software architectures. In David Wile, editor, *Proc. of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'94)*, pages 164–174. ACM Press, 1994.

[NR97] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In *ACM Sigsoft Software Engineering Notes. ESEC/FSE'97*, volume 22 no 6, November 1997.

[SG96] M. Shaw and D. Garlan. *Software Architecture, perspectives on an emerging discipline*. Prentice Hall, 1996.

[SL96] J. Souquières and N. Lévy. PROPLANE : A Specification Development Environment. In *Fifth Int. Conf. on Algebraic Methodology and Software Methodology (AMAST'96)*, volume 1101, Munich (G), July 1996. Lecture Notes in Computer Science.

[Tur96] K. J. Turner. Relating architecture and specification. *To appear in Computer Networks and ISDN Systems*, April 1996.

[Tur98] K. J. Turner. Validating architectural feature descriptions using LOTOS. In Kristofer Kimbler and Wiet Bouma, editors, *Proc. 5th. International Workshop on Feature Interactions in Telecommunication Networks and Software Systems*, pages 247–261. IOS Press, Amsterdam, Netherlands, September 1998.

6 A Summary of LOTOS used in this paper

LOTOS [BB87] is a formal specification language developed to specify open distributed systems. A LOTOS specification describes the global behavior of interacting processes. A process can be parameterized by abstract data types, and it can exchange typed values with other processes and call functions to transform data. Communication between processes in LOTOS is synchronous, i.e., two processes must participate in a common action at the same time. *Gates* are used to synchronize processes and to exchange data. Each process definition has the syntactic form

```
process process_name[gate_list](params): func:=
  behaviour behav_expr
  where local_def_list
endproc
```

where ***func*** indicates whether the process terminates (***func***`= exit` or `exit(v)` if it terminates sending a value `v`) or not (***func***`= noexit`). The behavior expression describes the sequences of observable actions that may occur at the gates of the process. Process definitions may include instantiations of processes.

The choice operator `[]` is used when alternative behaviors are allowed. The behavior expression `P1 [] P2` expresses that exactly one of the two processes will be executed, depending on a choice of the environment.

The behavior expression `P1 ||| P2` (interleaving) expresses that the two processes `P1` and `P2` behave independently and in parallel.

The behavior expression `P1[G] |[G]| P2[G]` (parallel composition) expresses that the two processes `P1` and `P2` must synchronize on the gate `G`. During the synchronization, they may exchange data. To synchronize, two processes must contain an action via the same gate `G`. To exchange data, one of them must contain an action `G ? t: v` which reads a value `v` of type `T` via gate `G`. The other process must contain an action `G ! exp` that writes a known value `exp` of type `T` onto the gate `G`. It is also possible to read or write more than one value in the same action, for example writing `G ? v: T ?w: T'`. An action can be guarded by a predicate, for example writing `G ? v: T [`*pred*`]`.

Behaviors may be made conditional by using the guard operator `[`*pred*`] ->` *beh* . The behavior expression *beh* will take place only if the predicate *pred* is satisfied.

In LOTOS, data are described using abstract data types with conditional equations and an initial semantics. Abstract data types are used for describing process parameters and values exchanged by the processes.